Math 365 / Comp 365: Homework 4

Charles Zhang

Note: Problems 2 through 4 on this homework were all taken from the take-home portions of old midterm exams.

```
require(Matrix)
require(jpeg)
set.seed(365)
```

Feel free to use my implementation of `vnorm` and `jacobi` and `GaussSeidel`:

```
# Vector norm
vnorm = function(v,p=2) {
  if ( p =="I") {
    return(max(abs(v)))
  }
  else {
    return(sum(abs(v)^p)^(1/p))
  }
}
```

```
# Solves Ax = b iteratively using the Jacobi Method
# m is the maximum number of iterations: default m = 25
# p is the p value of the matrix norm
# tol is the stopping tolerance (using the relative residual norm on the backward error)
jacobi = function(A,b,m=25,x = rep(0,n),p=2,tol=0.5*10^(-6),history=FALSE) {
  n = length(b)
  if (history) {
    hist = matrix(NA,nrow=length(b),ncol=(m+1))
    hist[,1] = x
  }
  d = diag(A)
  R = A
  R[cbind((1:n),(1:n))] = 0  # allows for r to be sparse
  steps=0
  for (j in 1:m) {
    x = (b - R %*% x)/d
    steps = steps+1
    if (history) {hist[,(j+1)] = as.matrix(x)}
    if (vnorm(b-A%*%x,p) <= vnorm(b,p)*tol) break
  }
  if (history) return(list(x=x,iterations=steps,history = hist[,1:(steps+1)]))
  else return(list(x=x,steps=steps))
}
```

```
GaussSeidel = function(A,b,m=25,x = rep(0,n),p=2,tol=0.5*10^(-6),history=FALSE) {
  n = length(b)
  if (history) {
    hist = matrix(NA,nrow=length(b),ncol=(m+1))
    hist[,1] = x
  }
  d = diag(A)
  L = A
```

```
  U = A
  U[lower.tri(A,diag=TRUE)] = 0  # U is the upper triangular part of A
  L[upper.tri(A,diag=TRUE)] = 0  # L is the lower triangular part of A
  steps=0
  for (j in 1:m) {
    bu = b - U %*% x
    steps = steps+1
    for (i in 1:n){
      x[i] = (bu[i] - L[i,] %*% x)/d[i]    # successivly update x as we use it.
    }
    if (history) {hist[,(j+1)] = as.matrix(x)}
    if (vnorm(b-A%*%x,p) <= vnorm(b,p)*tol) break
  }
  if (history) return(list(x=x,iterations=steps,history = hist[,1:(steps+1)]))
  else return(list(x=x,steps=steps))
}
```

Problem 1

a) Find (by hand but then check your work in R) the Cholesky factorization $A = R^T R$ of the matrix

$$A = \begin{pmatrix} 4 & -2 & 0 \\ -2 & 2 & -3 \\ 0 & -3 & 10 \end{pmatrix}.$$

Solution:

$$A = \begin{pmatrix} 4 & -2 & 0 \\ -2 & 2 & -3 \\ 0 & -3 & 10 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -3 \\ 0 & -3 & 10 \end{pmatrix} \begin{pmatrix} 2 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = R_1^T A_1 R_1$$

$$A_1 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -3 \\ 0 & -3 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{pmatrix} = R_2^T A_2 R_2 = R_2^T I R_2$$

$$\therefore A = R_1^T R_2^T R_2 R_1 = R^T R$$

$$\therefore R = R_2 R_1 = \begin{pmatrix} 2 & 0 & 0 \\ -1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 2 & -1 & 0 \\ 0 & 1 & -3 \\ 0 & 0 & 1 \end{pmatrix}$$

```
R = cbind(c(2,0,0), c(-1,1,0), c(0,-3,1))
(A = t(R) %*% R)
```

```
##      [,1] [,2] [,3]
## [1,]    4   -2    0
## [2,]   -2    2   -3
## [3,]    0   -3   10
```

b) Show that the Cholesky factorization procedure fails for the matrix

$$B = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 1 & 3 \\ 0 & 3 & 4 \end{pmatrix}.$$

$$B = \begin{pmatrix} 4 & 2 & 0 \\ 2 & 1 & 3 \\ 0 & 3 & 4 \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 \\ 1 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 0 & 3 \\ 0 & 3 & 4 \end{pmatrix} \begin{pmatrix} 2 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} = R_1^T B_1 R_1$$

The dignal of matrix B_1 contains 0 so that it can never become an indentity matrix. Therefore, the Cholesky factorization procedure fails for the matrix B.

c) Why does the Cholesky factorization work for the symmetric matrix A, but not for the symmetric matrix B? Hint: `eigen` may be useful.

```
eigen(A)$values
```

```
## [1] 11.05968527  4.86598780  0.07432693
```

```
B<-rbind(c(4,2,0), c(2,1,3), c(0,3,4))
eigen(B)$values
```

```
## [1]  6.405125  4.000000 -1.405125
```

Therrefore, only symmetric matrix which is positive definite like A can be Cholesky Factorized, and B can't.

Problem 2

Tridiagonal matrices show up a lot in applications. These are matrices whose only nonzero entries are on the main diagonal and just above or just below the main diagonal.

a) Write a function `TriDiag(d,a,b)` which takes three vectors as input and puts them on the 3 diagonals. For example, if you call

```
TriDiag( c(5,3,5,4,4), c(1,2,3,4), c(11,12,13,14) )
```

you should get the matrix:

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5   11    0    0    0
## [2,]    1    3   12    0    0
## [3,]    0    2    5   13    0
## [4,]    0    0    3    4   14
## [5,]    0    0    0    4    4
```

Try to use as few `for` loops as possible (you can actually do it without any `for` loops).

Bonus: Build in an option to output a sparse matrix (in which case the `Matrix` package is required). In practice, you would almost always be using sparse matrices when dealing with large, tridiaganal matrices.

```
TriDiag <- function(d,a,b, sparse=FALSE) {
  n <- length(d)
  if ( length(a) != (n-1) || length(b) != (n-1) )
    stop('vectors length diffeerent!')
  if (sparse) {
    Diag <- Matrix(0,nrow=n,ncol=n,sparse=TRUE)
    diag(Diag) <- d
  } else  Diag <- diag(d)
  Diag[cbind((1:(n-1)),(2:n))] <- a
  Diag[cbind((2:n),(1:(n-1)))] <- b
  return(Diag)
}
TriDiag(c(5,3,5,4,4), c(1,2,3,4), c(11,12,13,14))
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    5    1    0    0    0
## [2,]   11    3    2    0    0
```

```
## [3,]    0   12    5    3    0
## [4,]    0    0   13    4    4
## [5,]    0    0    0   14    4
```

```
TriDiag(c(5,3,5,4,4), c(1,2,3,4), c(11,12,13,14), TRUE)
```

```
## 5 x 5 sparse Matrix of class "dgCMatrix"
##
## [1,]  5  1  .  . .
## [2,] 11  3  2  . .
## [3,]  . 12  5  3 .
## [4,]  .  . 13  4 4
## [5,]  .  .  . 14 4
```

b) Illustrate your function on the following command:

```
n = 10
TriDiag(rep(5,n),rep(-1,n-1),rep(1,n-1))
```

```
##       [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
##  [1,]    5   -1    0    0    0    0    0    0    0     0
##  [2,]    1    5   -1    0    0    0    0    0    0     0
##  [3,]    0    1    5   -1    0    0    0    0    0     0
##  [4,]    0    0    1    5   -1    0    0    0    0     0
##  [5,]    0    0    0    1    5   -1    0    0    0     0
##  [6,]    0    0    0    0    1    5   -1    0    0     0
##  [7,]    0    0    0    0    0    1    5   -1    0     0
##  [8,]    0    0    0    0    0    0    1    5   -1     0
##  [9,]    0    0    0    0    0    0    0    1    5    -1
## [10,]    0    0    0    0    0    0    0    0    1     5
```

c) Use your `TriDiag` function and my implementation of Jacobi's method (above) to solve Computer Problem 2 in Section 2.5 of the book. Don't worry about finding the number of steps to reach a certain forward error. Instead, just try it with some different numbers of steps, and comment briefly on the convergence. Why might the convergence behave this way?

```
n <- 100
A <- TriDiag(rep(2,n),rep(1,n-1),rep(1,n-1))
b <- rep(0,n)
b[1] <- 1
b[n] <- -1

solution = solve(A,b)
sol1 = jacobi(A,b,p="I")
forward1 = vnorm(solution-sol1$x,"I")/vnorm(solution,"I")
forward1
```

```
## [1] 1
```

```
sol2 = jacobi(A,b,m=100,p="I")
forward2 = vnorm(solution-sol2$x,"I")/vnorm(solution,"I")
forward2
```

```
## [1] 0.9999994
```

```
sol3 = jacobi(A,b,m=1000,p="I")
forward3 = vnorm(solution-sol3$x,"I")/vnorm(solution,"I")
forward3
```

```
## [1] 0.7792718
```

```
sol4 = jacobi(A,b,m=10000,p="I")
forward4 = vnorm(solution-sol4$x,"I")/vnorm(solution,"I")
forward4
```

```
## [1] 0.01008219
```

```
D = diag(diag(A))
R = A-D
Dinv = solve(D)
eigen(-solve(D) %*% R)$values
```

```
##   [1]  0.99951628  0.99806560  0.99564935  0.99226987  0.98793044
##   [6]  0.98263525  0.97638942  0.96919900  0.96107094  0.95201311
##  [11]  0.94203426  0.93114406  0.91935304  0.90667260  0.89311502
##  [16]  0.87869340  0.86342170  0.84731470  0.83038798  0.81265791
##  [21]  0.79414165  0.77485711  0.75482294  0.73405853  0.71258396
##  [26]  0.69042002  0.66758814  0.64411041  0.62000955  0.59530886
##  [31]  0.57003226  0.54420418  0.51784962  0.49099408  0.46366353
##  [36]  0.43588442  0.40768361  0.37908840  0.35012645  0.32082577
##  [41]  0.29121471  0.26132192  0.23117632  0.20080707  0.17024356
##  [46]  0.13951534  0.10865215  0.07768385  0.04664039  0.01555181
##  [51] -0.01555181 -0.04664039 -0.07768385 -0.10865215 -0.13951534
##  [56] -0.17024356 -0.20080707 -0.23117632 -0.26132192 -0.29121471
##  [61] -0.32082577 -0.35012645 -0.37908840 -0.40768361 -0.43588442
##  [66] -0.46366353 -0.49099408 -0.51784962 -0.54420418 -0.57003226
##  [71] -0.59530886 -0.62000955 -0.64411041 -0.66758814 -0.69042002
##  [76] -0.71258396 -0.73405853 -0.75482294 -0.77485711 -0.79414165
##  [81] -0.81265791 -0.83038798 -0.84731470 -0.86342170 -0.87869340
##  [86] -0.89311502 -0.90667260 -0.91935304 -0.93114406 -0.94203426
##  [91] -0.95201311 -0.96107094 -0.96919900 -0.97638942 -0.98263525
##  [96] -0.98793044 -0.99226987 -0.99564935 -0.99806560 -0.99951628
```

The forward error will converge to 0, but the convergence rate is very slow as the steps change significantly but error only changes a little. It will converge because the eignevalues of matrix $-D^{-1}R$ are less than one, where D is the diagnal matrix of A, and R = A-D. It converges slowly because many magnitudes of eigenvalues are very closed to 1.

Problem 3: Sparse LU Factorization

Sparse matrices commonly appear when trying to solve partial differential equations by discretizing them. There are different methods to do this, but one of the most straightforward uses finite difference approximations for the partial differential operators, just as you used finite difference approximations for the derivative operator in Technical Report 1.

For example, when doing a discrete approximation to a 2D heat diffusion problem, you end up with a sparse matrix with a block structure. Specifically, if the approximation grid contains k_1 by k_2 interior locations, you end up with a $(k_1 \cdot k_2) \times (k_1 \cdot k_2)$ matrix of the form

$$A = \begin{bmatrix} B & -I & & & & \\ -I & B & -I & & & \\ & -I & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & -I \\ & & & & -I & B \end{bmatrix},$$

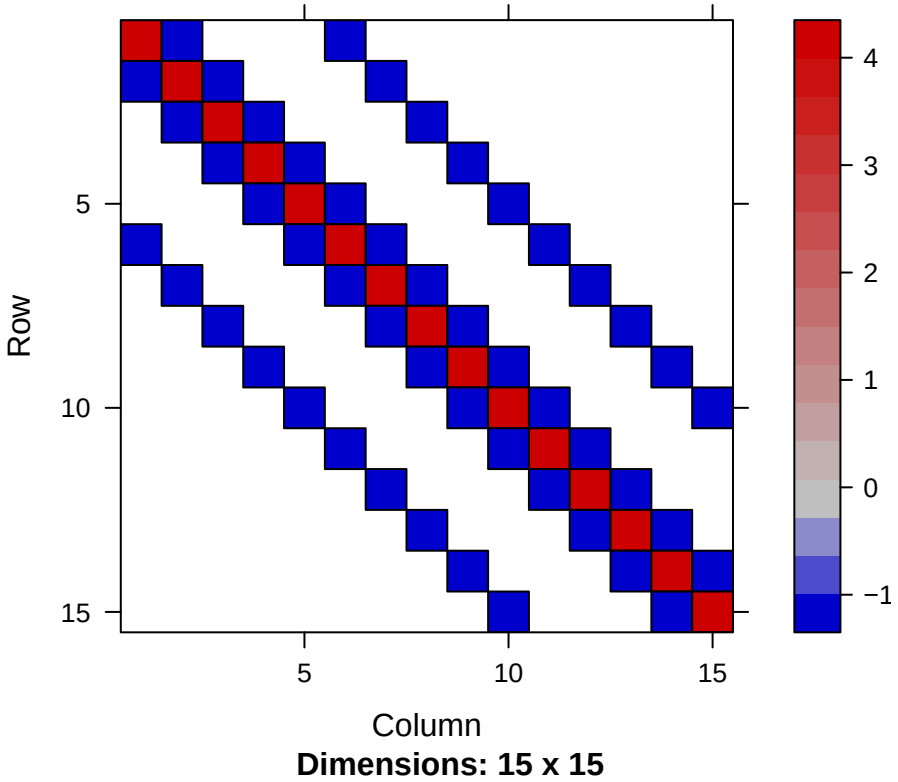
where each I is the $k_1 \times k_1$ identity matrix, and each B is a $k_1 \times k_1$ matrix of the form

$$B = \begin{bmatrix} 4 & -1 & & & & \\ -1 & 4 & -1 & & & \\ & -1 & \ddots & \ddots & & \\ & & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & -1 \\ & & & & -1 & 4 \end{bmatrix}.$$

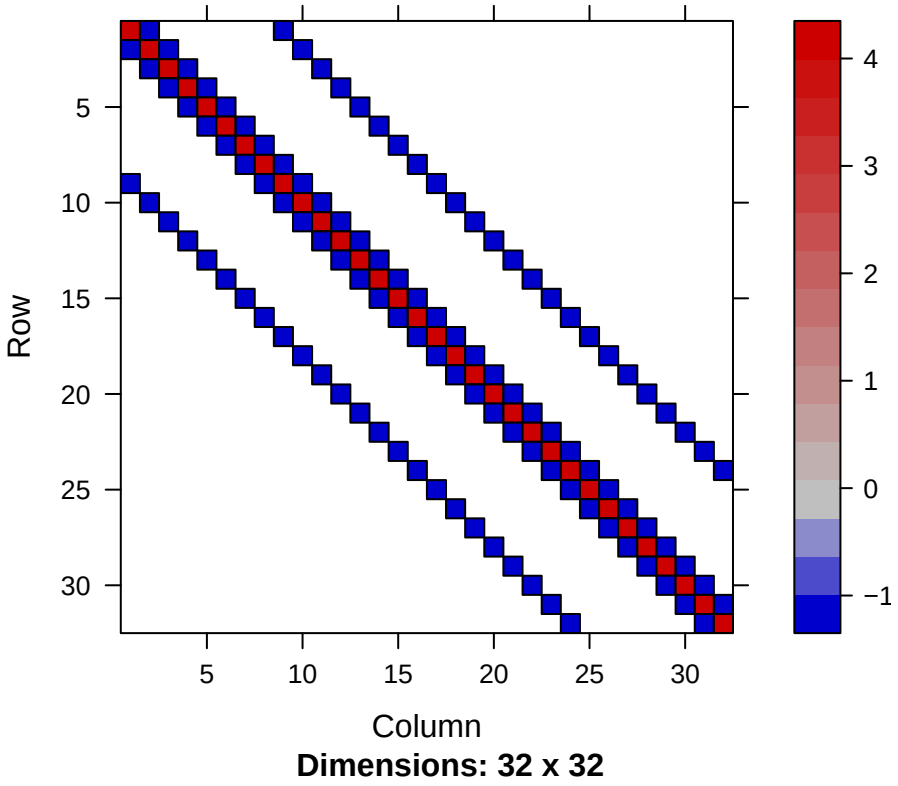
(a) Generate the matrix A above for the case of $k_1 = 5$ and $k_2 = 3$.

***Extra Credit*: Write a function `sparsePattern(k1,k2)` that takes any values of k_1 and k_2 and returns the matrix A. Check that it matches the examples above and try it on a couple of different matrices.**

```
sparsePattern = function(k1, k2){
  N = k1*k2
  i = c((k1+1):N, 1:(N-k1), 2:N,1:(N-1), 1:N, seq(k1+1, N, by=k1), seq(k1,(N-1),by=k1))
  j = c(1:(N-k1), (k1+1):N,1:(N-1), 2:N, 1:N, seq(k1, (N-1), by=k1), seq(k1+1,N,by=k1))
  x = c(-rep(1,2*(N-k1)), -rep(1,2*(N-1)), rep(4,N), rep(0,2*(k2-1)))
  A = sparseMatrix(i=i, j=j, x=x)
return(A)
}

A=sparsePattern(5,3)
image(A)
```

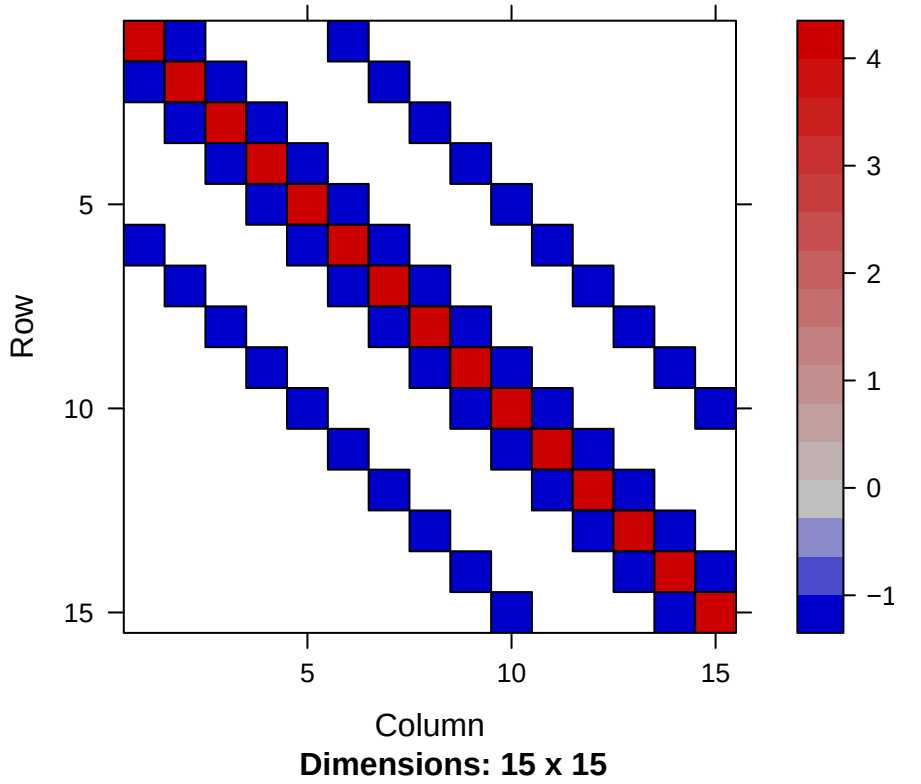


```
A=sparsePattern(8,4)
image(A)
```



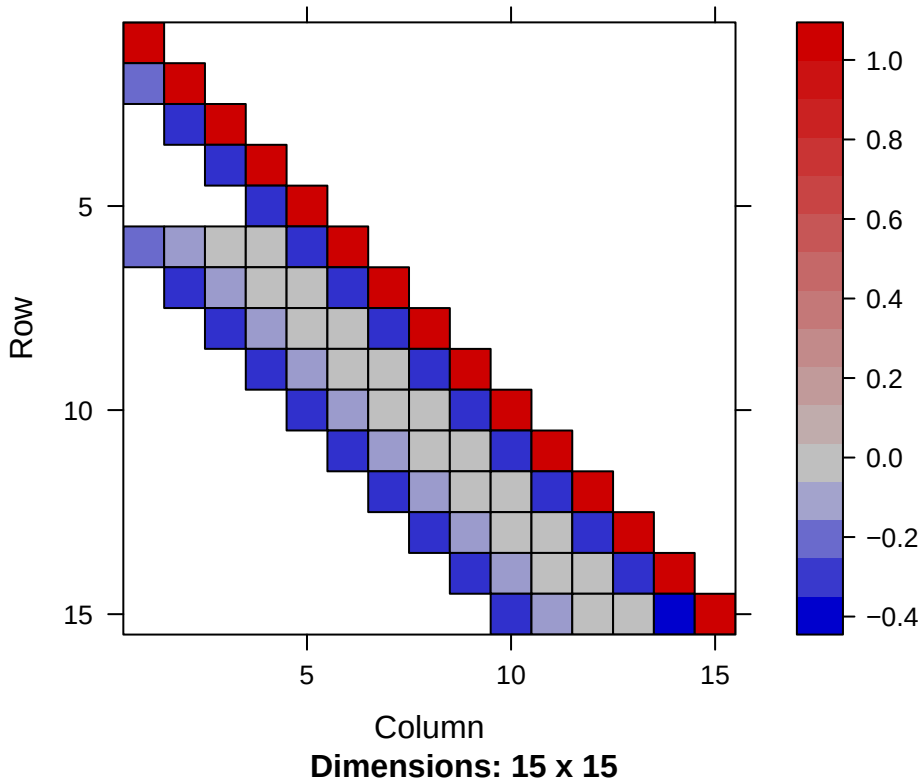
One issue with using the LU decomposition on a sparse matrix like A is that the factors L and U have more non-zero entries than A. This phenomenon is called *fill-in*. Here is an example you can try once you have formed A:

```
myLUFast = function(A,tol=10^-8) {
  n = nrow(A)
  L = diag(x=1,nrow=n)  # start with L = identity matrix
  U=A
  for ( k in 1:(n-1) ) {
    pivot = U[k,k]
    if (abs(pivot) < tol) stop('zero pivot encountered')
    mults=U[(k+1):n,k]/pivot
    U[(k+1):n,k]=0
    U[(k+1):n,(k+1):n]=U[(k+1):n,(k+1):n]-mults%o%U[k,(k+1):n]
    L[(k+1):n,k]=mults
  }
  return(list(L=L,U=U))
}
A=sparsePattern(5,3)
out=myLUFast(as.matrix(A))
L=as(out$L,"sparseMatrix")
U=as(out$U,"sparseMatrix")
image(A)
```



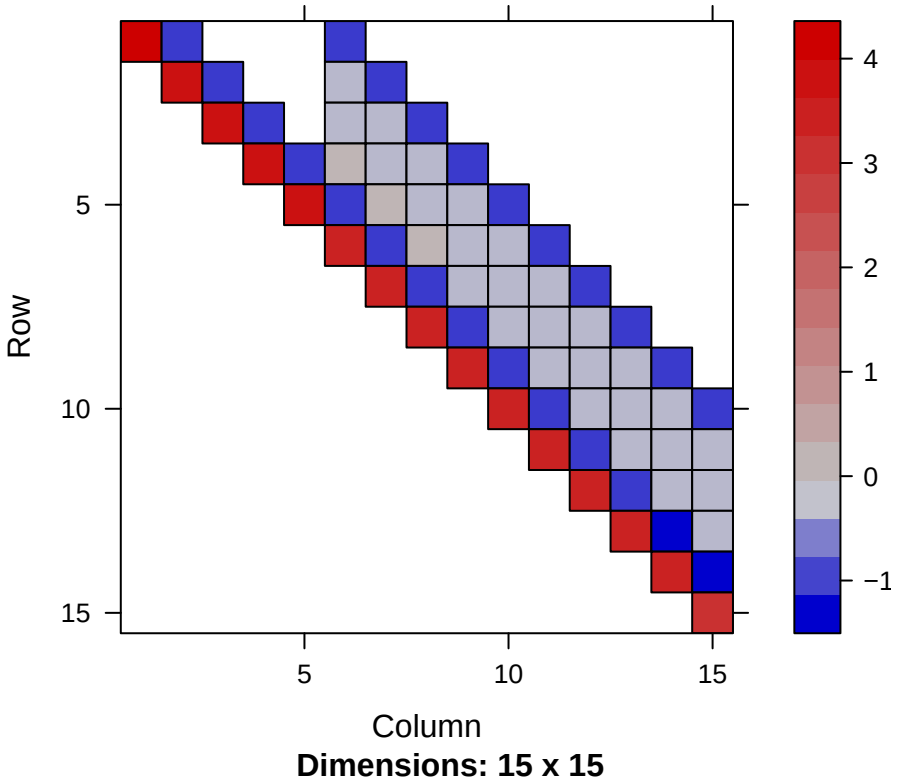
Dimensions: 15 x 15

```
image(L)
```



Dimensions: 15 x 15

```
image(U)
```



Can we find triangular matrices L and U that are sparser and still satisfy $A = LU$? No, recall that the factorization is unique. Instead, one approach is to force L and U to be sparser, but have LU only approximate A.

b) Your main task is to write a function `mySparseLU` that returns a lower unit triangular matrix L and an upper triangular matrix U such that L and U only have off-diagonal non-zeros entries in the locations where A has off-diagonal non-zero entries. In the following pseudocode for the algorithm, $NZ(A)$ is the set of indices of non-zero entries of A. For example, if $A_{3,4} \neq 0$, then $(3,4) \in NZ(A)$.

Pseudocode for Sparse LU Approximate Factorization

Input: A

Output: L (lower unit triangular) and U (upper triangular)

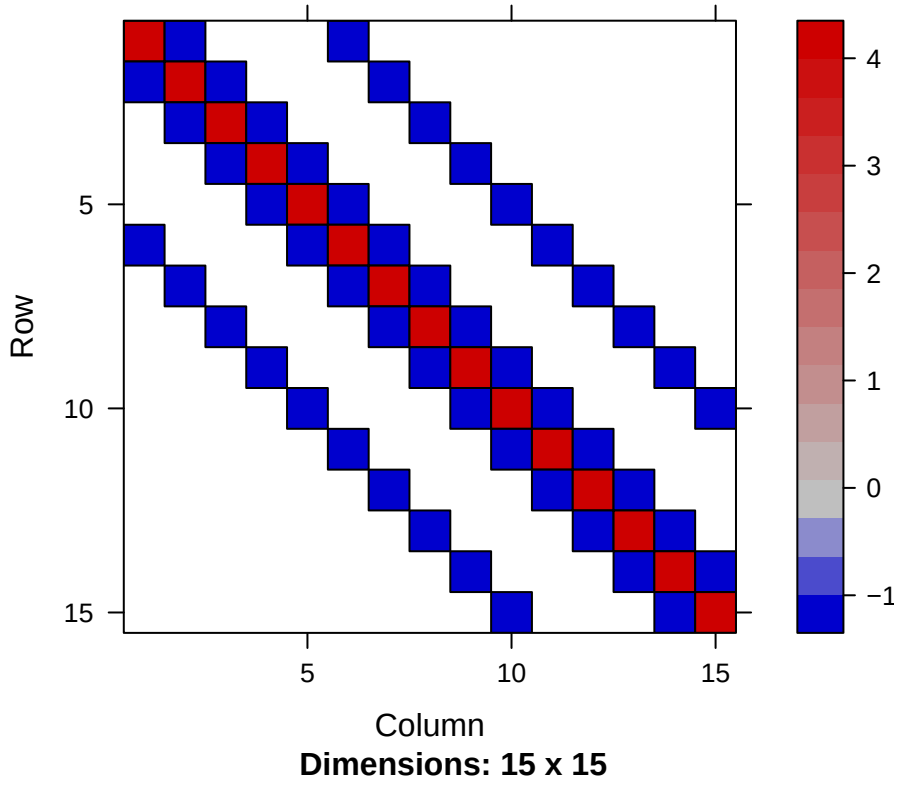
1. **for** $i = 2, 3, \ldots, n$ **do**
2. **for** $k = 1, 2, \ldots, i-1$ and for $(i,k) \in NZ(A)$ **do**
3. $A_{ik} = A_{ik}$ / A_{kk}
4. **for** $j = k+1, \ldots, n$ and for $(i,j) \in NZ(A)$ **do**
5. $A_{ij} = A_{ij} - A_{ik}A_{kj}$
6. **end for**
7. **end for**
8. **end for**
9. Let U be the upper triangular part of A (including the diagonal)
10. Let L be the strictly lower triangular part of A (not including the diagonal)
11. Set every element of the diagonal of L to be 1

You may find the functions `upper.tri` and `lower.tri` useful. Your function `mySparseLU(A)` should return a list containing L and U.

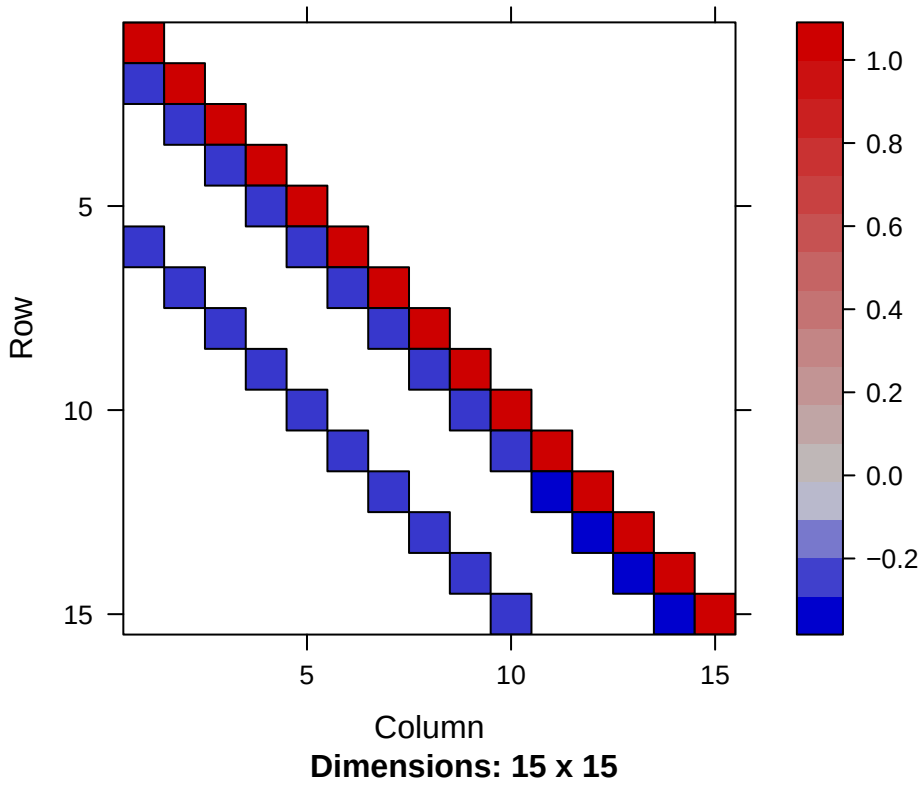
```
mySparseLU = function(A){
  n = nrow(A)
  for (i in (2:n)){
    for (k in (1:(i-1))){
      if (A[i,k]!=0){
        A[i,k]<-A[i,k]/A[k,k]
        for (j in ((k+1):n)){
          if (A[i,j]!=0){
            A[i,j]<-A[i,j]-A[i,k]*A[k,j]
          }
        }
      }
    }
  }

  U = A
  U[lower.tri(U)] = 0

  L = A
  L[upper.tri(L)] = 0
  diag(L) = rep(1,n)

  return(list(L=L,U=U))
}
```

To test your function, use your A matrix from part (a), with $k_1 = 5$ and $k_2 = 3$, and double check that your L and U matrices have the same sparsity pattern as the lower and upper triangular parts of the original A, respectively:

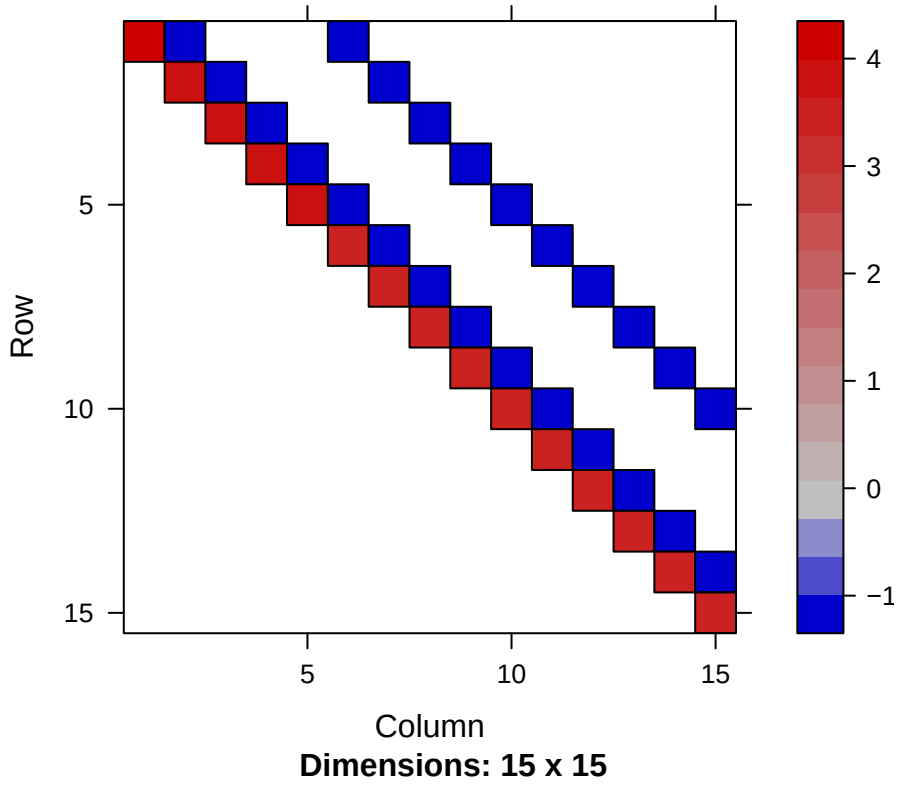
```
out=mySparseLU(A)
image(A)
```


Row
Column
Dimensions: 15 x 15
5
10
15
4
3
2
1
0
−1

```
image(out$L)
```


Row
Column
Dimensions: 15 x 15
5
10
15
1.0
0.8
0.6
0.4
0.2
0.0
−0.2

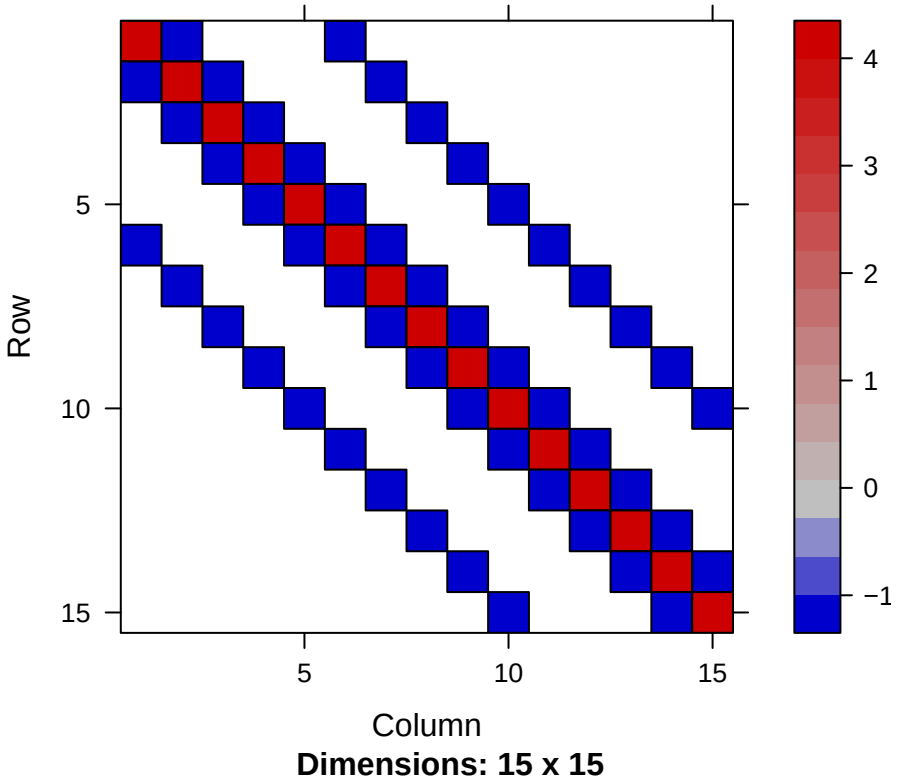
```
image(out$U)
```



Dimensions: 15 x 15

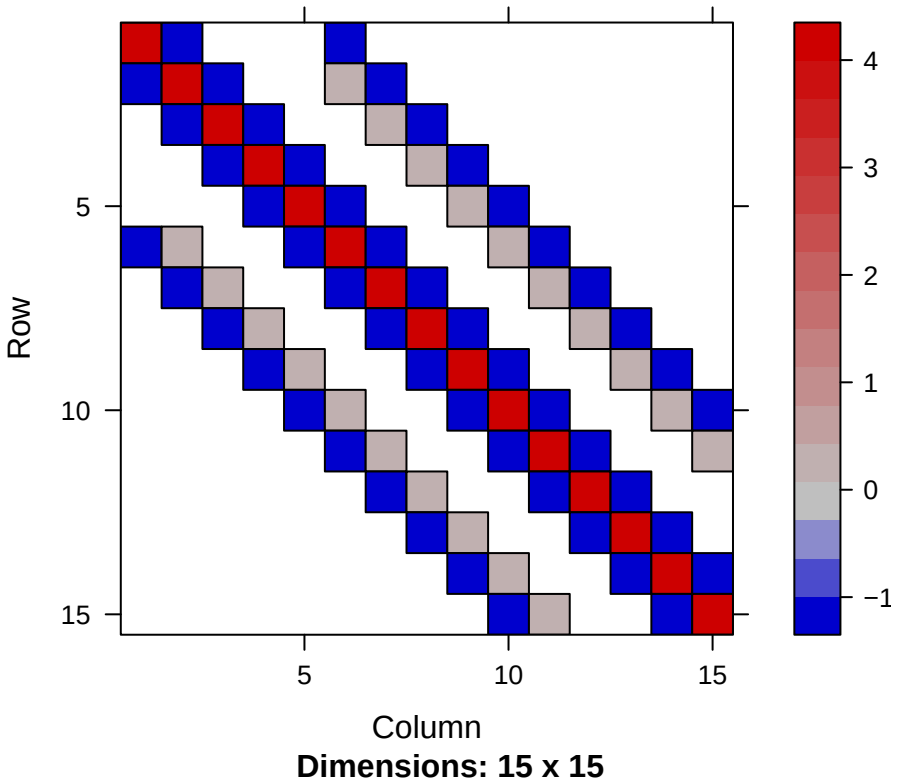
While it is nice to preserve the sparsity pattern, we can also see that A is only approximately equal to LU:

```
approx=out$L%*%out$U
approx.error=A-approx
image(A)
```



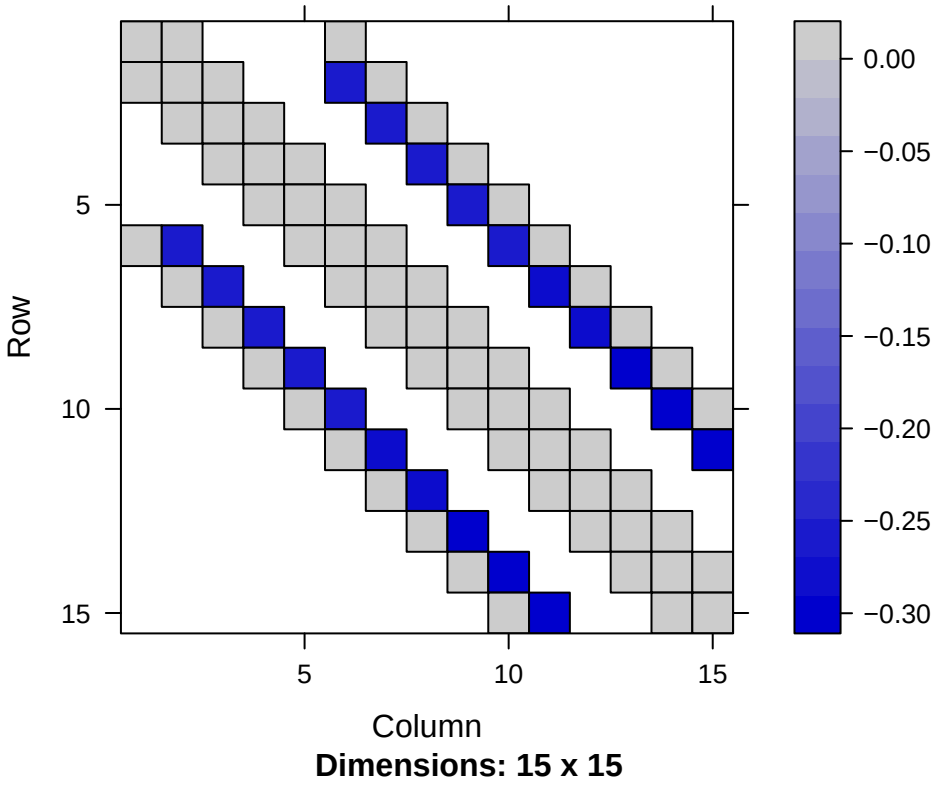
Dimensions: 15 x 15

```
image(approx)
```



Dimensions: 15 x 15

```
image(approx.error)
```



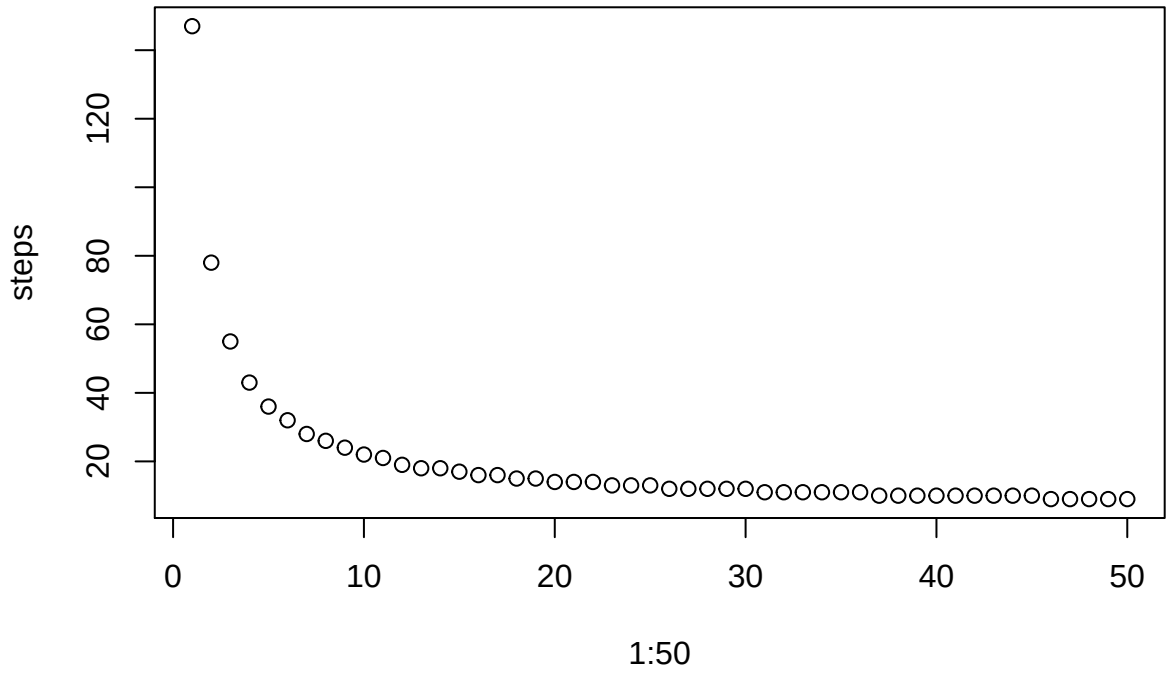
There is a tradeoff between how many additional non-zero entries we allow in L and U versus how accurately LU approximates A. Generalizations of the algorithm you implemented manage that tradeoff, allowing the user some choice over how accurate of an approximation she requires for the application at hand.

Problem 4: Diagonal Dominance and Jacobi Convergence

Perform a numerical experiment (or experiments) to test the following statement: *In general, the greater the diagonal dominance in a matrix, the faster the Jacobi method converges to the solution.* I am leaving it completely open ended as to how you do this. Be sure to explain your work and precisely how your numerical results support your conclusions.

```
A = matrix(rexp(100, rate=1), ncol=10)
b <- runif(n = 10, min = 1, max = 10)
Steps = function(A, b, d) {
  steps = vector()
  for (j in 1:length(d)) {
    diag(A) = 0
    for (i in 1:10) {
      A[i,i] = sum(A[i,]) + d[j]
    }
    steps = c(steps, jacobi(A,b, m=100000, tol=10e-8)$steps)
  }
  return(steps)
}

steps = Steps(A, b, 1:50)
```

```r
plot(1:50, steps)
```



I randomly created a 10× 10 matrix A and length 10 vector b. By chaning the diagonal entry to the sum of that row adding 1 to 50, matrix A is more and more diagonal dominant. Saving the steps that Jacobi method generated to converge to the solution, with the tolerance = 10e-8, we can see in the graph above that steps decrease significantly with greater diagonal dominance. Therefore, I can conclude that in general, the greater the diagonal dominance in a matrix, the faster the Jacobi method converges to the solution.

Problem 5: Jacobi and Gauss-Seidel Methods

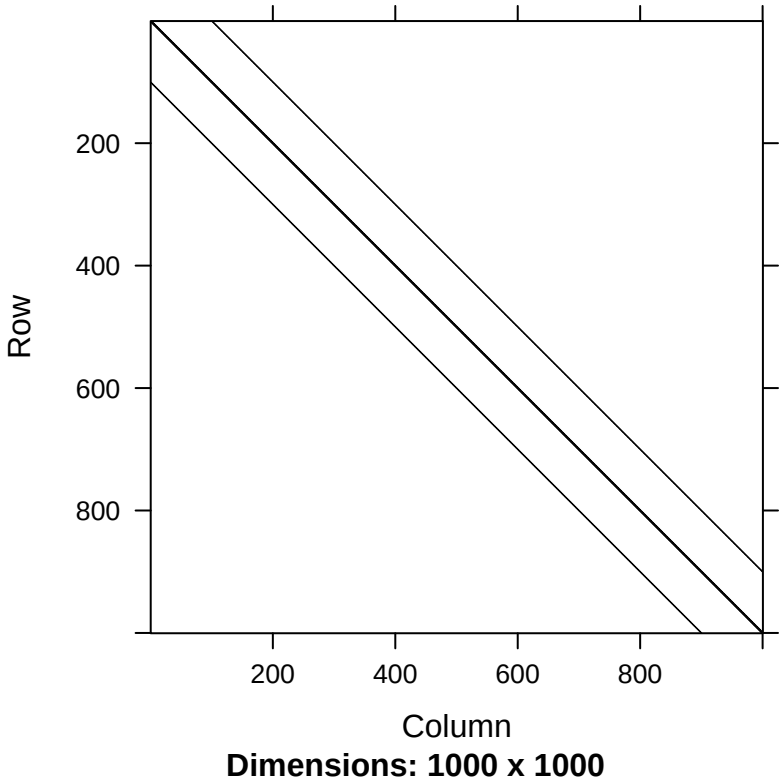
This problem is from the activity on iterative methods.

We are interested in solving our favorite problem: $Ax = b$, where b is a 1000×1 column vector with every entry equal to 1. For A, we will use the `ThreeBanded` function we wrote in the activity on sparse matrices with $N = 1000$ and an offset of 100.

```r
ThreeBanded=function(n,offset){
  spMatrix(n, n,
           i=c(1:n, 1:(n-1), 2:n, (offset+1):n, 1:(n-offset)),
           j=c(1:n, 2:n, 1:(n-1), 1:(n-offset), (offset+1):n),
           x=c(.5+sqrt(1:n), rep(1,(2*(2*n-1-offset)))))
}

n=1000
A=ThreeBanded(n,100)
```

Let's inspect our A matrix to see the sparsity pattern:

```
image(A)
```



We'll use two different iterative methods to solve $Ax = b$: Jacobi's method and Gauss-Seidel.

Feel free to use my implementation of `vnorm` and `jacobi` and `GaussSeidel`:

a) Briefly discuss if Jacobi's method will converge on this matrix A and why.

It converges beacuse the Jacobi method converges if the matrix A is strictly diagonally dominant.

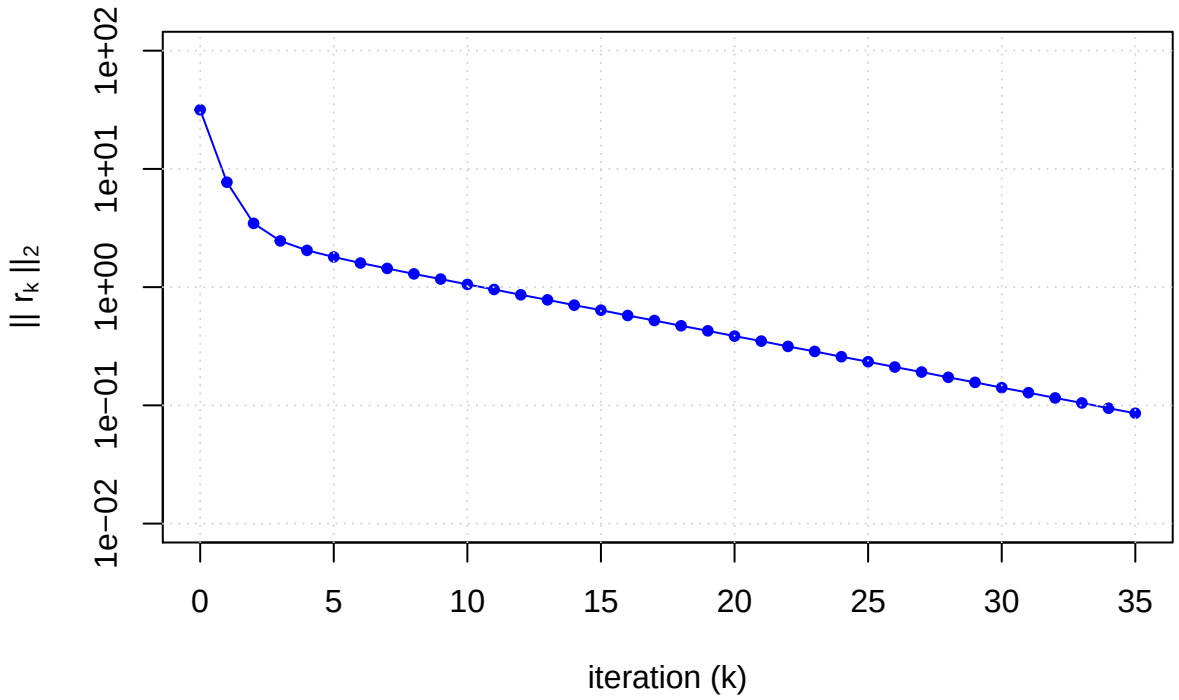
b) Run 35 iterations of Jacobi's method, with an initial guess, $x^{(0)}$, of all zeros. From the sequence of approximations $\{x^{(k)}\}_{k=0,1,\ldots,35}$, compute the sequence of residuals $\{r^{(k)}\}_{k=0,1,\ldots,35}$ with $r^{(k)} = b - Ax^{(k)}$, and then compute the two-norms of the residual vectors. Save your residual norms in a vector called `jac.res.norms`, and plot the residual norms versus the iteration number using the following code:

```
plot(0:35,jac.res.norms,pch=20,type="o",log="y",
     xlab="iteration (k)",
     ylab=expression("|| r"[k]~"||"[2]),
     col="blue",ylim=c(1e-2,1e2))
grid()
```

Note: to compute the residuals, you will either have to modify the output of my `jacobi.r` code or compute the residuals after calling that function. Either is fine, but I recommend the latter method, as it will make the subsequent part of this problem easier.

```
b = matrix(1,nrow=1000,ncol=1)

out = jacobi(A,b,m=35,history=TRUE)
res = rep(b,nrow=n)-A %*% out$history
jac.res.norms = sqrt(colSums(res*res))
plot(0:35,jac.res.norms,pch=20,type="o",log="y",
```

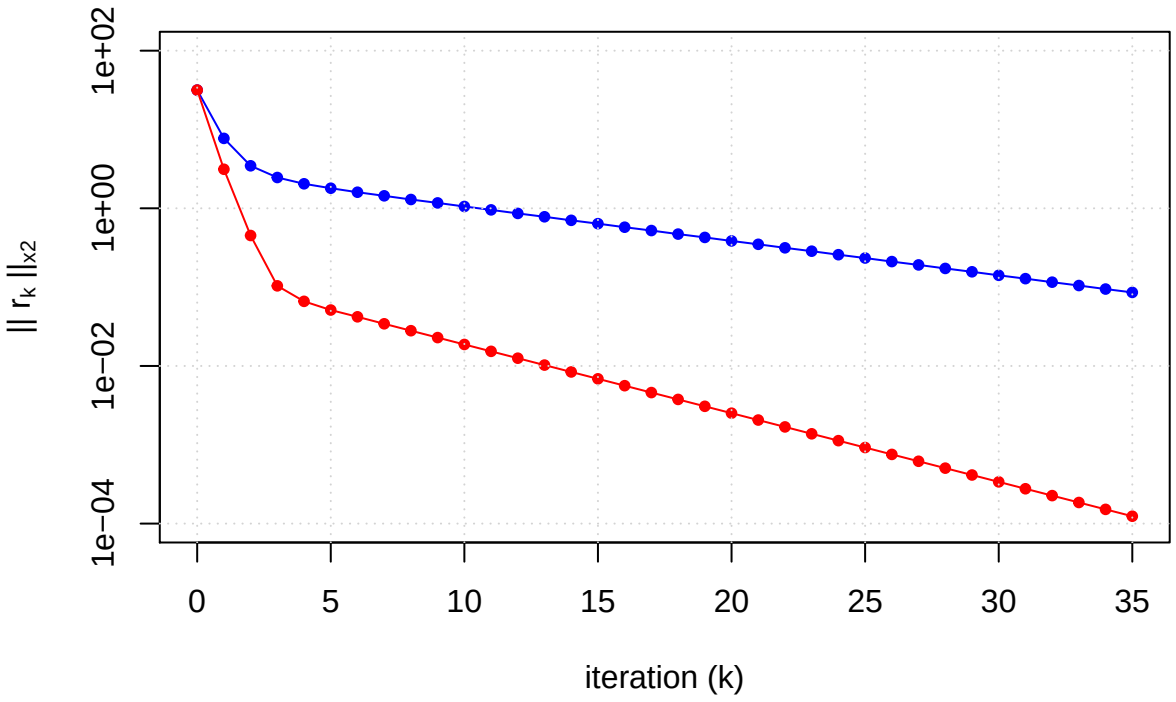
```
     xlab="iteration (k)",
     ylab=expression("|| r"[k]~"||"[2]),
     col="blue",ylim=c(1e-2,1e2))
grid()
```



c) Now, repeat the process with the Gauss-Seidel method, saving the residual norms as `gs.res.norms`, and plot using the following code:

```
plot(0:35,jac.res.norms,pch=20,type="o",log="y",
    xlab="iteration (k)",
    ylab=expression("|| r"[k]~"||"[2]),
    col="blue",ylim=c(1e-4,1e2))
lines(0:35,gs.res.norms,pch=20,type="o",col="red")
grid()
```

```
gs.out=GaussSeidel(A, b, m=35, history = TRUE)
gs.res=rep(b, (35+1), nrow=n) - A %*% gs.out$history
gs.res.norms=sqrt(colSums(gs.res*gs.res))
plot(0:35,jac.res.norms,pch=20,type="o",log="y",
     xlab="iteration (k)",
     ylab=expression("|| r"[k]~"||"[x2]),
     col="blue",ylim=c(1e-4,1e2))
lines(0:35,gs.res.norms,pch=20,type="o",col="red")
grid()
```



Problem 6: Easy linear solvers

We have discussed methods for solving $Ax = b$, but I claim there are a few matrices which can be solved "easily". Write programs to solve these five kinds of "easy" systems of equations (most only require a line or two of code, and the more difficult ones you have seen before). See the comments in the model codes below for details of what input arguments each function should take. Once you have written each program, demonstrate it by solving the example matrix I have included in each question. In commented lines, I have provided code that sets up the problem. Once your program works, you can just uncomment these lines. *Finally, state what the computational complexity is of each method for A an $n \times n$ matrix (no justification is required).*

a) Diagonal Matrices

$$A = \begin{pmatrix} 2 & & \\ & 3 & \\ & & -0.5 \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ 6 \\ 5 \end{pmatrix}$$

```
my.solve.diagonal <- function(d,b){
  # Solve the system D x = b
  # The full matrix D is not provided. Instead, we have d,
  #    which is a vector (not a matrix) containing the diagonal
  #    entries of D.  This allows us to avoid storing lots of zeros.

  # You add code here building x from d and b.
  x = c(b/d)
  return(x)
}

x <- my.solve.diagonal(c(2,3,-0.5),c(3,6,5))
print(x)
```

```
## [1]   1.5   2.0 -10.0
```

Time Complexity: O(N)

b) Upper Triangular Matrices

$$A = \begin{pmatrix} 2 & 1 & -1 \\ & 3 & 2 \\ & & -0.5 \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ 7 \\ 5 \end{pmatrix}$$

```
my.solve.uppertri <- function(U,b){
  # Solve the system Ux = b
  # assume that U is a full matrix with zeros below the diagonal.

  # You add code here building x using U and b.
  n = nrow(U)
  sol = rep(NaN, n)
  sol[n] = b[n] / U[n, n]
  for (i in (n-1):1){
    sol[i] = (b[i] - sum(sol[(i+1):n] %*% U[i, (i+1): n]))/U[i, i]
  }
  return(sol)
}

U <- cbind(c(2,0,0),c(1,3,0),c(-1,2,-0.5))
x <- my.solve.uppertri(U,c(3,7,5))
print(x)
```

```
## [1]  -8   9 -10
```

Time Complexity: $O(N^2)$

c) Lower Triangular Matrices

$$A = \begin{pmatrix} 2 & & \\ 0 & 3 & \\ -1 & 3 & -0.5 \end{pmatrix}; \quad b = \begin{pmatrix} 4 \\ 6 \\ 5 \end{pmatrix}$$

```
my.solve.lowertri <- function(L,b){
  # Solve the system Lx = b
  # assume that L is a full matrix with zeros above the diagonal.

  # You add code here building x using L and b.
  n = nrow(U)
  sol = rep(NaN, n)
  sol[1] = b[1] / U[1, 1]
  for (i in 2:n){
    sol[i] = (b[i] - sum(sol[0: (i-1)] %*% U[i, (i-1): 0]))/U[i, i]
  }

  return(sol)
}

L <- cbind(c(2,0,-1),c(0,3,3),c(0,0,-0.5))
x <- my.solve.lowertri(L,c(4,6,5))
print(x)
```

```
## [1]   2   2 -10
```

Time Complexity: $O(N^2)$

d) Orthogonal Matrices

$$A = \begin{pmatrix} \frac{-1}{9} & \frac{8}{9} & \frac{4}{9} \\ \frac{8}{9} & \frac{-1}{9} & \frac{4}{9} \\ \frac{4}{9} & \frac{4}{9} & \frac{-7}{9} \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ 6 \\ 9 \end{pmatrix}$$

```
my.solve.orthogonal <- function(Q,b){
  # Solve the system Qx = b
  # assume that Q is a full orthogonal matrix

  # You add code here building x using Q and b.
  x = t(Q )%*% b
  return(x)
}
Q <- cbind(c(-1,8,4),c(8,-1,4),c(4,4,-7))/9
x <- my.solve.orthogonal(Q,c(3,6,9))
print(x)
```

```
##      [,1]
## [1,]    9
## [2,]    6
## [3,]   -3
```

Time Complexity: $O(N^2)$

e) Permutation Matrices

$$A = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}; \quad b = \begin{pmatrix} 3 \\ 6 \\ 5 \\ 1 \end{pmatrix}$$

```
my.solve.permutation <- function(p,b){
  # Solve the system P x = b
  # note: a full matrix P is not provided.  Instead we have
  #    a vector of integers called p.  This gives the position of the 1 in each row.
  # for example, if p = (2,3,1,4), ...
  #   then P = rbind(c(0,1,0,0),c(0,0,1,0),c(1,0,0,0),c(0,0,0,1)).

  # You add code here building x such that Px=b.
  x[p] = b[1:length(p)]
  return(x)
}

p <- c(2,4,1,3)
x <- my.solve.permutation(p,c(3,6,5,1))
print(x)
```

```
## [1] 5 3 1 6
```

Time complexity: O(N)